Environmental Health Perspectives
Vol. 87, pp. 5–11, 1990

Managing Clinical Research Data: Software Tools for Hypothesis Exploration

by C. F. Starmer* and M. A. Dietz*

Data representation, data file specification, and the communication of data between software systems are playing increasingly important roles in clinical data management. This paper describes the concept of a self-documenting file that contains annotations or comments that aid visual inspection of the data file. We describe access of data from annotated files and illustrate data analysis with a few examples derived from the UNIX operating environment. Use of annotated files provides the investigator with both a useful representation of the primary data and a repository of comments that describe some of the context surrounding data capture.

Introduction

The nature of the computational platform on which experimentally derived data are managed and analyzed can play an important role in the quality of a study. For instance, the platform can provide no services to the user, thus forcing a large manual and potentially error-prone effort in data acquisition, preparation, management, and analysis. On the other hand, the platform can provide a wide variety of ancillary services, from basic data handling and quality assessment to data subsetting and value transformation to model building, parameter estimation, hypothesis testing, and data visualization. A long-term problem within the computer science community has been that of developing effective, easy-to-use interfaces between the user and these ancillary services. There have been two traditional approaches: a static, compiled specification of the data management process and an adaptive, interpretive specification of the process.

The computational platform could facilitate matching the needs of the investigator with the nature of his/her investigation and the translation of these needs into easily evoked actions. Our approach to research data management follows the adaptive, interpretive paradigm and is based on two observations: it is extremely difficult to design a detailed data management and analysis protocol prior to the onset of the investigation, and the specification of data management actions such as editing and analysis seems better suited to visual specification (with icons and selection with a pointing device such as a mouse) than to a particular command language. We often find that as the project unfolds, new unexpected observations are made that influence the outcome attributes to be monitored, their frequency, and associated covariates. Furthermore, as initial hypotheses are tested, new or refined hypotheses may develop. To try to match the evolutionary nature of an investigation with computer-based analytical tools, we have embarked on a program that emphasizes the ease of adaptation, evolution, and specification of data-manipulation protocols. This paper presents three main ideas: data representation, data file specification, and the interface between files and processes.

Questions arise when developing a data analysis procedure: Was the data collection protocol followed? Have the data been properly coded? Is the record format correct? Have the proper subsets been identified? The answers to some of these questions reside in documents distinct from the data files, and in the heads of personnel responsible for data preparation, data entry, and data file management. Is it possible to make progress with coordinating access to these various sources of information by changing the way we traditionally view the data capture-management-analysis process? Typically, these processes are executed and managed in a rigid and semiautonomous manner. That is, procedures are designed for each part of a study that are then carried out by different members of the investigative team. Consequently, important procedural information used for data capture might be deleted from primary data files after initial use and thus be unavailable for subsequent review. The rationale for pruning such data in the setting of the economics of today's computers and data storage devices is probably no longer valid.

We have been developing some guidelines and software tools for research data management and data analysis that focus on capturing and managing not only the primary data but also some of the experimental context surrounding data capture that was alluded to

*Duke Medical Center, Durham, NC 27710.
Address reprint requests to C. F. Starmer, P.O. Box 3181, Duke Medical Center, Durham, NC 27710.

previously. Our rationale is that it is simply unrealistic to expect to be able to explicitly state each step in the data management/analysis process correctly the first time, i.e., to specify the experimental and data capture protocols, data management protocols, and analytical procedures in sufficient detail that the process is mechanical. Rather, we accept that this process, from experiment to analysis, is evolutionary in nature and defies precise specification. The efficiency of the analysis can depend critically on allowing the primary investigator the flexibility of navigating freely from tentative analyses that raise questions about experimental protocol to the raw primary data in order to investigate anomalies. If such navigation is permitted, then anything that can aid the investigator with the visual review of primary data and results of analyses would be helpful. This stepwise, incremental refinement or evolution of the experimental and analytical procedures is becoming more commonplace as we build more flexibility in the data management systems. Access to inexpensive highspeed processors and bulk storage have made this approach both feasible and cost effective.

This paper focuses on three primary areas: data representation, data file specification that incorporates protocol information into the primary files, and standards of communication between files and data transformation/analysis procedures that facilitate reuse of analysis tools. We also show how these issues have a direct impact on the flexibility or plasticity of a data management system.

Viewing an Experiment as an Object

We start by viewing the experiment as an object or entity that consists of experimental protocols, recorded data, and analysis procedures. We view the entire object as the subject of analysis, not just the recorded response data. This viewpoint reflects the fact that the recorded data includes both signals as well as noise, and the noise can sometimes represent important hidden signals reflecting protocol weaknesses or distortion introduced by a data-capture instrument. Sometimes, this hidden signal goes unrecognized until some preliminary analyses have been completed. Thus, hypothesis testing in such a setting frequently involves exploring a class of alternate hypotheses. When results of preliminary analyses introduce questions that reflect uncertainty about protocols or instrumentation, it is useful to review protocols, sources of signals, instrumentation, and potential confounding factors, and to make adjustments to and refinement of the region of hypothesis exploration.

To facilitate user-directed refinement, we have chosen to represent all aspects of the study including experimental protocols, data, and analysis output as ASCII records that comprise a data file (Fig. 1). This figure shows a segment of a typical data file derived from the study of interactions between a drug and a cell membrane. Shown is the dictionary record that names the variables stored in data records (record 1, leading character is a $) followed by comment (leading character is a !) records that describe the type of experiment, date, instrumentation settings and data tape identifier followed by a frame identifier (leading character is a :) that

```
$event Ipeak I_min t_min
!Propoxyphene study; 50 μM concentration 10/12/88b Recovery and
  uptake
!protocols. Filter  0  5  0  × 100
!Follower gain = x2
!Tape 208. speed = 15 ips
!Sampling rate = 20 kHz, 2 channels, 600 points/frame
:train .25 Hz
407     1.3625          - .4            0.0011
408     1.2225          - 0.095         0.0009
409     1.245           - 0.05          0.0008
410     1.2075          - .4625         0.001
411     1.145           - 0.0525        0.0009
412     1.1775          - 0.05          0.0009
413     1.2             - 0.095         0.001
414     1.17            - 0.0625        0.0009
415     1.1575          - 0.0525        0.0008
:train .5 Hz
438     1.2475          - .17           0.001
439     1.1625          - 0.055         0.0008
440     1.05            - 0.045         0.0008
441     1.0025          - .24           0.001
442     1               - .325          0.0011
443     .985            - .135          0.0009
444     .9675           - .4225         0.0013
445     .955            - .1675         0.0009
446     .98             - .15           0.001
447     .945            - .1375         0.001
448     .9525           - .4225         0.0011
449     .945            - 0.05          0.0008
:train 1 Hz
469     1.21            - .19           0.0009
470     1               - .415          0.0012
471     .865            - .24           0.0009
472     .8275           - .1            0.001
473     .7825           - 0.0475        0.001
474     .795            - 0.0425        0.0009
475     .75             - .2275         0.001
476     .7425           - 0.0475        0.0009
477     .7375           - .2075         0.001
478     .78             - 0.0875        0.001
479     .7325           - .2925         0.001
:train 2 Hz
500     1.19            - .44           0.001
501     .9125           - 0.0325        0.0008
502     .7325           - 0.04          0.001
503     .64             - .3175         0.0011
504     .6125           - .3875         0.0013
505     .535            - 0.035         0.0008
506     .58             - 0.05          0.001
507     .515            - 0.0325        0.0007
508     .5175           - .13           0.0009
509     .52             - .32           0.001
510     .4975           - 0.0325        0.0007
```

FIGURE 1. An annotated file. The first record (line) is a dictionary record (identified by $ in the first position) that identifies the four columns of data as event, Ipeak, I_min and t_min. The next five lines are comment records (identified by a ! in the first position) that describe the experiment and other important information. The next record initiated by the colon, :, indicates a frame of data defined as the next nine records. This is followed by another frame identifier and a group of data records. Each frame of data is identified by a text string that can be used to extract the frame using the function, select.

states the protocol associated with the following data records. Though the advantages of capturing and making accessible these data are obvious to many, few have explored ways to access and manipulate all these elements in a manner that eases the burden of interpreting data and analysis results. By using an ASCII representation of all characterizations of a study, the annotation and data records can be readily printed and displayed with full screen editors, and they can perhaps be manipulated by tools designed to perform simple data extraction or data transformation tasks. With properly designed data-manipulation tools, it should rarely be necessary to write a program to explicitly display or transform data. To facilitate the manipulation of these experimental objects, we have developed a standard file format that we call an annotated file.

Data File Structure

To provide the investigator with the widest possible territory for exploration, data files should capture as much of the recorded data and experimental context as possible. Thus, we have defined data files in terms of two components: actual data records that reflect attribute values characterizing responses and base-line properties of experimental units and annotation or comment records that capture the context associated with the data record and give the files a self explanatory flavor when viewed. Traditional data records and annotation records are distinguished by using a reserved character in the first position of the record.

Data records are composed of alphanumeric fields of text separated by white space. Unknown values are represented by dk (don't know). Among the annotation records, we include a dictionary record that names each field of data (e.g., event, peak current (Ipeak), minimum current (I_min), time to peak (t_min) etc.); frame separators that separate groups of data records that are logically related (e.g., different frames reflecting different experimental conditions); and comment records that can appear anywhere in the file and contain text that helps the reader visually interpret data in the region of the comment, as seen in Figure 1.

A formal description of a file is shown in Figure 2, where each construct is expressed in Backus Normal Form. The file definition includes that of a classical flat file so that nonannotated files can be manipulated with the same ease as annotated files, and it provides backward compatibility with old files. For nonannotated files, the variables are assumed to be named x1, x2....

Annotated File Access

To manipulate data, estimate model parameters, or summarize results, it must be easy to access records from the data file, independent of whether the record is a data record, dictionary, comment, or frame record. In order to remove this burden from each analysis procedure, we have built a small library (RECORDIO) of file access procedures (Table 1) that can be called by analysis procedures in order to access data.

File rules

‹file›	:=	‹simple frame› | ‹dictionary›‹frames›
‹frames›	:=	‹frames›‹frame› | ‹frame›
‹frame›	:=	‹label›‹simple frame›
‹simple frame›	:=	‹dictionary›‹datablock› | ‹datablock›
‹datablock›	:=	‹dataline› | ‹datablock›‹dataline› | NULL
‹dataline›	:=	‹comment› | ‹data›

Record rules

‹dictionary›	:=	$‹fields›‹cr›
‹label›	:=	:‹string›‹cr›
‹comment›	:=	!‹string›‹cr›
‹data›	:=	‹separator›‹fields›‹cr› | ‹fields›‹cr›
‹separator›	:=	‹separator›‹sep› | ‹sep›
‹sep›	:=	space | tab
‹cr›	:=	carriage return
‹fields›	:=	‹field› | ‹fields›‹separator›‹field›
‹string›	:=	‹char› | ‹string›‹char›
‹char›	:=	any printable ASCII character
‹field›	:=	‹char_field› | ‹number› | ‹dk›
‹dk›	:=	dk | ?
‹number›	:=	any number recognized by scanf
‹char_field›	:=	"‹string›" | ‹special_string›
‹special_string›	:=	a string of printable ASCII characters that is not exactly "dk", and does not include space, tab, comma, or question mark

FIGURE 2. BNF specification of an annotated file. This specification describes in recursive terms each construct in an annotated file. The construct <a> := <b> is read a is defined by b. The construct <a> | <b> is read a or b. The construct <a><b> is read a and b. Primitive constructs such as alphanumeric characters are not enclosed in <>. For instance, file is defined as a simple frame or a dictionary and frames. A simple frame is defined as a dictionary and a data block or a data block. A data block is defined as a data line or a data block and a data line.

We have defined four record types: DATA, COMMENT, DICTIONARY, and FRAME. The RECORDIO library of procedures contains modules that either can return each data record independent of record type or can read records sequentially and ignore records of a certain type. As an example of a procedure that returns a record independent of its type

get_record (buffer)

gets the next record from the file, returns the text of the record in the array, buffer, and returns the record type as the value of get_record. For numerical procedures, one can use the procedure

get_vector (vector,*m*)

to read the next DATA record while ignoring any annotation records. This procedure gets the next record of type, DATA, interprets up to m data fields returning their values in vector[ ], and returning the number of fields actually found or End of File as the value of the function. Thus the C code segment

```
double vector [MAX];
int m;
while (get_vector (vector, m) ! = EOF)
{
    process values in vector[i]
}
```

Table 1. RECORDIO library.

Procedure name	Function
get_record (buffer)	Gets the next record from the standard input device, stores as a text string in buffer, and returns the record type or end of file as the value of the function.
fget_record (fp,buffer)	Same as get_record except input is specified by fp (a file pointer).
get_fields (fields)	Same as get_record except each field of the record is identified and pointed by an element of the vector, fields[ ].
get_vector (values, *m*)	Gets the next DATA record and stores the value of each of *m* fields as elements of the vector, value. The returned value of get_vector is the number of fields converted or end of file.
fget_vector (fp,values,*m*)	Same as get_vector except input is specified by fp.
sget_vector (buffer,value *m*)	Same as get_vector except the input is contained in the character string, buffer.
get_vvector (value,valid)	Gets the next DATA record and stores the value of each field in the vector, value, and stores the value 1 in the corresponding element of valid if the field is numerical and 0 if it is blank specification, dk.

will read and process each data record, while ignoring each comment embedded in the file.

If it is important to reproduce the annotation records in the output data stream, one can use the procedure, get_record(buffer) to access the next record, and then if its type is not DATA, print the record, else use sget _vector(buffer,vector,m) to fetch the data values from the text string, buffer. Here, the code segment previously given is modified by reading each record with get_record( ):

```
char buffer [LENGTH]
double vector [MAX];
int type,m;
while ((type = get_record(buffer)) ! = EOF)
{
      switch (type)
      {
            COMMENT:
            DICTIONARY:
            SEPARATOR:
                  print("%s\n",buffer);
                  break;
            DATA:
                  sget_vector
                  (buffer,vector,m);
                  process values of vector[i]
                  break;
      }
}
```

To select a group of data records that are bounded by frame separators, we have constructed a procedure, select "identifying test," that reads each record from a data file. This procedure searches for a frame separator (indicated by a leading :) followed by the text specified as identifying text and passes to the output stream all records until the next frame separator (or End of File) is encountered. Thus to extract "train 1 Hz" one would specify (Fig. 3):

```
select <input.dat "train 1 Hz" | process_data
```

Here the | represents the UNIX pipe command, and it means that the output of the left member is passed as input to the right member. The < symbol identifies the input data file.

In addition we have a tool, doall "command string" that manipulates each frame of data in the same manner according to the command sequence specified by "command string." To fit an exponential to the x,y values represented by the first two fields of each group of records (frame) labeled "train" is specified by

```
select <input.dat "train" | doall "tf x1 x2 | fitexpc"
```

where tf extracts the first two fields, x1 and x2, from each record and passes the resultant values to a procedure that fits an exponential using column 1 as the dependent variable and column 2 as the independent variable. If fitexpc requires the independent variable first followed by the dependent variable, one could use tf to reorder the values, as with tf x2 x1.

For data transformation, we have developed a tool, tf, that can perform simple data transformations as specified on the command line. If the input data stream contains a dictionary record, then the data transformation is specified using the variables named within the dictionary record. Files without a dictionary record use the default names, x1, x2... as previously described. Thus, to access the values of the second and fourth column of the file, we specify:

```
tf < filename x2 x4
```

If the file contains a dictionary as in Figure 1, we would specify:

```
tf <subset·dat Ipeak t_min
```

If one wanted to evaluate an algebraic transformation, one would specify for example

```
tf <subset·dat event_new_variable
      = Ipeak*sqrt(t_min)
```

Sometimes the order of variables as they appear in the data file must be modified. To exchange the positions of event and Ipeak, one would specify:

```
tf <subset·dat Ipeak event
```

Analysis

UNIX and many microcomputer operating systems such as DOS treat data files and keyboard data in a

```
rm2000% select ‹subset.dat "train 1 Hz"
$event Ipeak I_min t_min
!Propoxyphene study; 50 μM concentration 10/12/88b Recovery and
  uptake
!protocols. Filter 0  5  0  × 100
!Follower gain = ×2
!Tape 208, speed = 15 ips
!Sampling rate = 20 kHz, 2 channels, 600 points / frame
:train 1 Hz
469          1.21          -  .19          0.0009
470          1             -  .415         0.0012
471          .865          -  .24          0.0009
472          .8275         -  .1           0.001
473          .7825         - 0.0475        0.001
474          .795          - 0.0425        0.0009
475          .75           -  .2275        0.001
476          .7425         - 0.0475        0.0009
477          .7375         -  .2075        0.001
478          .73           - 0.0875        0.001
479          .7325         -  .2925        0.001
rm2000% select ‹subset.dat "train 1 Hz" | tf Ipeak
$×1
! tf commands -> Ipeak
! Old Dictionary -> $event Ipeak I_min t_min
!Propoxyphene study; 50 μM concentration 10/12/88b Recovery and
  uptake
!protocols. Filter 0  5  0  ×100
!Follower gain = ×2
!Tape 208, speed = 15 ips
!Sampling rate = 20 kHz, 2 channels, 600 points / frame
:train 1 Hz
1.21
1
 .865
 .8275
 .7825
 .795
 .75
 .7425
 .7375
 .73
 .7325
rm2000% select ‹subset.dat "train 1 Hz" | tf Ipeak | fitexpc -x
n = 11
Exponential curve fit: a + b*exp(-c*x)
Residual std. dev.0.014277
Iterations = 7
a = 0.737361 sd(a) = 0.007028
b = 0.470020 sd(b) = 0.014448
c = 0.583496 sd(c) = 0.041921

 x            y          expected(y)
0.000000      1.210000      1.207381
1.000000      1.000000      0.999606
2.000000      0.865000      0.883679
3.000000      0.827500      0.818999
4.000000      0.782500      0.782910
5.000000      0.795000      0.762775
6.000000      0.750000      0.751541
7.000000      0.742500      0.745273
8.000000      0.737500      0.741776
9.000000      0.730000      0.739824
```

FIGURE 3. Partial output an analysis of the data in Figure 1. Here an exponential was fit to the second column of data (Ipeak) from the frame labeled "train 1 Hz." The string, rm2000%, is the prompt from the computer and the following text is the command line typed by the user. The text between prompts is computer output. Here, the process, select, extracted the appropriate frame of data as illustrated at the top on the figure. This frame then had its second column extracted by the process, tf, and the resultant data passed to fitexpc for fitting. The process was initiated by the command line: select <subset·dat "train 1 Hz" | tf Ipeak | fitexpc -x".

uniform, device-independent, and interchangeable manner. The transparency of the data source within the UNIX or the DOS operating environment can be used very effectively for data analysis. Further, the pipe command, |, that allows processes to be concatenated where the output of the first process is treated as input to a successor process, provides a powerful tool for specifying an analysis procedure.

Instead of building a large, multifunction program for data analysis, one can more easily construct an equivalent procedure by using a composition of single-function analysis tools and specify the composition as a pipe. This encourages the reuse of single function pieces of software, since many times the same analysis program can be used in several different settings. The major difficulty encountered with this approach is that of separating experimental data from control data, e.g., the specification of analysis options. This is because the analysis program should not be required to know anything about input data formatting. Traditional batch processing systems often forced one to mix procedure commands with primary data, thereby forcing the analysis program to know when to expect data and when to expect procedure information or analysis options. Here, we have chosen to separate the primary data stream from the control data stream by requiring the specification of analysis options as text on the command line and only allowing observational (or annotated) data to flow through the input and output stream. As an example,

```
fitexp <input.dat -a 5.0 -c 2.5
```

requests fitting an exponential of the form y = a*exp(-c*x) where initial values of a and c are 5.0 and 2.5. Similarly,

```
linmodel <input.dat -m 3 -0
```

requests fitting a linear model of three elements (-m 3) to the data in input·dat and to include a mean term (-0) in the model; i.e., $y = a_0 + a_1x_1 + a_2x_2$. The command line

```
linmodel -m 3
```

fits a model of the form $y = a_0x_0 + a_1x_1 + a_2x_2$

The central idea here is to develop statistical analysis procedures as simple filters where output results are generated from primary data that is accessed from the standard input device and control or procedure specific information that is accessed from the command line. Using the data file access modules listed in Table 1, individual analysis procedures can be coded that accept data from annotated files. The output of the program can be in the traditional form of output, or output can appear with annotation records embedded in the output data stream.

As an example, we can specify fitting an exponential to the second column (Ipeak) in the data file illustrated in Figure 1 where Ipeak is the dependent variable and the record number (0,1,2...) is the independent vari-

able. (The independent variable values are generated by the procedure as requested by the −x option on the command line.) Using the fitting procedure, fitexpc which fits functions of the form

$$y = a + b*\exp(-c*x),$$

the command line appears as

```
select <subset·dat "train" | doall "tf Ipeak | fitexpc −x"
```

which takes data from the file, subset·dat, selects each frame of records identified by "train" and executes the command line

```
tf Ipeak | fitexpc −x
```

for each frame. Here tf extracts the dependent variable (Ipeak) and pipes the sequence of values to fitexpc which generates a value of x (record number) for each value of y (specified by the −x option). The output appears in Figure 4.

If one is interested in only the estimated values of the exponential rate (c), they can be extracted by the UNIX function, grep, (get regular expression, print) which prints each line of its input that matches the requested string. Thus, grep "c =" extracts each line of input containing the string, c =. Figure 4 illustrates both this command line and the results. Note that the frame identifiers are reproduced in the output so that the results are still annotated and easy to interpret.

Discussion

As the capacity and speed of desktop computers has increased, the rate-limiting step for some data analysis activities has moved from the machinery to the end user. It is appropriate then to ask what can be done to improve the quality of the interface between the investigator and the primary data, and hopefully achieve more thoughtful analyses by offering a friendly and flexible analysis environment.

We have identified three areas where the user interface can be significantly improved: readability of data files, communication between files and analysis procedures, and visualization of each step in an analysis procedure. To improve readability of files, we have found that much is gained when all data are represented in a uniform, unencoded fashion, such as with ASCII text. Furthermore, we have found that understanding the data values embedded in a data file is further enhanced if text comments are embedded in the files such that the context of the data records is maintained for visual inspection in the data file. Thus, not only is the file readable, but one is able to reconstruct a portion of the experimental protocol from which the data were acquired, removing potential ambiguities associated with complex data collection protocols.

Communication between data files and programs is facilitated by using the data stream transparency afforded by UNIX and DOS, i.e., data from files and from

```
rm2000% select ‹subset.dat "train" | doall "tf Ipeak | fitexpc −x"
:train .25 Hz
n = 9
Exponential curve fit: a + b*exp(−c*x)
Residual std. dev.0.028383
Iterations = 12
a = 1.173049 sd(a) = 0.014506
b = 0.182521 sd(b) = 0.030171
c = 0.804288 sd(c) = 0.321078

 x            y            expected(y)
0.000000      1.362500      1.355570
1.000000      1.222500      1.254710
2.000000      1.245000      1.209584
3.000000      1.207500      1.189395
4.000000      1.145000      1.180362
5.000000      1.177500      1.176321
6.000000      1.200000      1.174513
7.000000      1.170000      1.173704
8.000000      1.157500      1.173342
:train .5 Hz
n = 12
Exponential curve fit: a + b*exp(−c*x)
Residual std. dev.0.015165
Iterations = 5
a = 0.950815 sd(a) = 0.007584
b = 0.304199 sd(b) = 0.014914
c = 0.490303 sd(c) = 0.056425

 x            y            expected(y)
0.000000      1.247500      1.255014
1.000000      1.162500      1.137119
2.000000      1.050000      1.064915
3.000000      1.002500      1.020694
4.000000      1.000000      0.993612
5.000000      0.985000      0.977025
6.000000      0.967500      0.966867
rm2000% select ‹subset.dat "train"|doall "tf Ipeak|fitexpc −x|grep 'c ='"
  :train .25 Hz
c = 0.804288 sd(c) = 0.321078
:train .5 Hz
c = 0.490303 sd(c) = 0.056425
:train 1 Hz
c = 0.583496 sd(c) = 0.041921
:train 2 Hz
c = 0.534144 sd(c) = 0.038571
```

FIGURE 4. Further data extraction. Here the process, doall, fits an exponential to each frame of data that contains the string "train" and the UNIX function, grep (get regular expression print) is used to extract each line of output that contains the text "c =" which contains the rate constant estimated by the fitting procedure. This command line provides a batch processing facility when the same sequence of data manipulation steps is required for each frame.

the keyboard appear the same. Further improvement results from separating data from control information required by a particular analysis program. Following these two ideas, one can create a complex process by specifying a sequence of primitive processing steps where the output of each elementary process is the input for its successor process. This approach improves the reusability of individual processing elements and avoids the construction of large multistep processing programs that enjoy little reuse.

Finally, we believe much is to be done in the area of

visualization of data and analyses. This paper begins to explore some tools that allow one to display data in tabular or graphical form and cut and paste segments that can feed an analysis procedure. It is in this area of visual analysis that we believe significant progress can be made in removing insulation that traditionally shields the investigator from the primary data.

This research was supported in part by grants HL11307 and HL32994 from the National Institutes of Health and contract 4414804 from the Office of Naval Research.